Indirect Data Representation Via Offset Vectoring: A Code-integrity-driven In-memory Data Regeneration Scheme

Erik Sonnleitner, Marc Kurz and Alexander Palmanshofer
Department for Mobility & Energy, University of Applied Sciences Upper Austria, Campus Hagenberg, Austria

Keywords: Code Security, Credential Storage, Steganography, Information Hiding.

Abstract: A common problem in software development is how to handle sensitive information required for appropriate process execution, especially when requesting user input like passwords or -phrases for proper encryption is not applicable due to I/O, UI or UX limitations. This often leads to such information being either stored directly in the source code of the application, or as plaintext in a separate file. We therefore propose an experimental scheme for dynamically recovering arbitrary chunks of information based on the integrity of the text-segment of a running process, without the information being easily extractible from either an on-disk binary, memory dump or the memory map of a running process. Implementing an algorithm we call *offset vectoring*, this method can help dealing with sensitive information and enhancing the resistance against attacks which aim at extracting such data as well as attempts towards modifying an application, e.g. for the purposes of cracking software.

1 INTRODUCTION

1.1 Handling Sensitive Data

In many cases, sensitive information (e.g. authentication credentials, cryptographic keys, etc.) is stored directly in an application's source code. Although bad practice, this happens very often and even for major brands Netgear (e.g. PSV-2018-0099), Cisco (e.g. CVE-2018-0222, CVE-2018-0268, CVE-2018-0271), Lenovo (e.g. CVE-2016-1491), HP (e.g. CVE-2015-2903) and numerous others.

This may be the case either if an application needs to authenticate to another service via a fixed key, or as a way to validate a given key or secret.

This makes it easy for an attacker to extract such information without even running the application, simply by analyzing the corresponding binaries using most simple and fundamental tools provided by most Unix-like operating systems like `strings(1)` or `objdump(1)`.

Moreover, tools like *Gitrob* (Henriksen, 2015) were created, being specifically designed for crawling through source-code platforms in order to search for sensitive information like hard-coded passwords and cryptographic private keys.

1.2 Memory Forensics

However, even in cases where strong cryptography is used to encrypt such information, there may be situations in which implementations will show security defects under certain conditions.

In order to given an illustrative example, consider a password manager as application. A security-aware password management software will strongly rely on using cryptography for securing password information, in most cases based on a user-provided passphrase, representing the master password for decrypting the password database.

Most of the time, e.g. in current implementations of major browsers like Mozilla Firefox, the master password will be asked for either upon start of process execution or the first time a stored password is requested. If no additional precautions are met (e.g. using browser extensions like *Master password timeout*[1]), the master password will be stored in-memory in plain-text upon decryption, and remain there until the application is terminated (Naaktgeboren, 2015).

Subsequently, if an attacker manages to acquire a snapshot of the current main memory process space, it's easily possible to extract the master pass-

[1]See https://addons.mozilla.org/en-us/firefox/addon/master-password-timeout/

Sonnleitner, E., Kurz, M. and Palmanshofer, A.
Indirect Data Representation Via Offset Vectoring: A Code-integrity-driven In-memory Data Regeneration Scheme.
DOI: 10.5220/0007786703330340
In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications (ICETE 2019)*, pages 333-340
ISBN: 978-989-758-378-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

word and hence all others using methods of memory forensics (Ligh et al., 2014). Acquiring memory snapshots has proven not to be too difficult after all, e.g. via accessing the main memory through `/proc/kcore`, accessing the process memory space through `/proc/<pid>/mem`, creating software plugins, forcing the application to dump core for further analisis, or via more exotic methods like side-channel attacks (Samyde et al., 2002) or (Zhenghong and Ruby, 2007).

1.3 Terminology

The terminology used throughout the entire document is listed in Table 1. In addition, whenever a new symbol is introduced, it is also explicitly outlined in the text of the corresponding section.

Table 1: Terminology.

Symbol	Definition
π_{ts},π_{te}	Start/end pointer in `.text` segment of the process
π_c	Pointer to current byte in segment iteration loop
$[\pi_c]$	Dereferencing the π_c pointer
β	Byte for XORing π_c contents
ν, $\|\nu\|$	Offset byte-vector & its size
σ	Key byte-vector
ω	Offset calculation for π_c in σ
θ	Initial offset added to π_{ts}
Θ	Optional array of initialization bytes (initialization vector)

2 RELATED RESEARCH

A related technique which tries to resemble certain information is used in the area of application exploitation, and is called *return-oriented programming* (Prandini and Ramilli, 2012) (ROP). The ROP approach was introduced after all major operating systems and CPU manufacturers started to enforce the well known `W^X` approach, shortly outlined in section 3.1. Hereby, memory pages are either writable or executable, but never both. In early days, when stack and heap were both writable and executable, the primary approach to acquire malicious code execution was to store shell code on the respective memory areas, and redirect function return addresses to somewhere near the shell code. Even if that address isn't known exactly, techniques like *NOP sleds* can diminish such shortcomings.

As a result of `W^X`, this isn't possible anymore. The ROP approach on the other hand, searches the `.text` section of a binary to find *gadgets*, short blocks of machine code with the last assembly operation being a `ret` statement. Using this technique, an attacker can search for multiple gadgets he or she needs for executing whatever needed, e.g. to prepare registers for a `system()` or `execve()` function call.

Regarding the use of credential information in applications, the CERN Computer Security Team has published guidelines how to deal with such situations (CERN Computer Security Team, 2015). In short, the best way is always to encrypt such information, in cases where a key is already present or can be acquired. However, if this is not possible, useful alternatives are sparse. Hereby, CERN recommends to externalize credentials to a separate file or a database. Both options might not fit well for certain situations, especially if the credentials are needed prior to connecting to a database, or if the credentials should not be stored in a plain-text file on the device. Even then, these strategies would not hinder an attacker to analyze externally stored information, often with even less effort.

The *offset vectoring* scheme presented in this paper uses the essential concept of the well-known ROP technique and utilizes it in order to recreate sensitive data based on such gadgets.

3 PREREQUISITES

In order to explain our proposed concept, some of the fundamental core concepts of how Linux handles process memory and how common modern memory protection techniques interfere with or modify these concepts need to be outlined, since they have a great impact on the implementation and inner workings of our proposal. Therefore, the following sections 3.1 to 3.3 discuss relevant prerequisites concerning (i) process memory layout, (ii) addressing memory protection schemes, and (iii) determining `.text` section start and end addresses.

3.1 Process Memory Layout

The memory layout of a running process under Linux consists of multiple *sections* or *segments*. Most notably, these sections are `.text` for actual machine code, `.data` for initialized global or static variables, `.bss` for non-initialized global or static data, `.heap` for dynamically run-time allocated memory, and `.stack` for local variables (Erickson, 2007).

Only heap and stack are dynamic in size. Moreover, most modern operating systems enforce the

`W^X` approach: Any section is either writable or executable, but never both. Depending on the operating system, this technique may also be called *Data Execution Prevention* (DEP) or *Stack NX/Heap NX*, and was introduced for memory protection in order to prevent exploits from executing previously introduced shell code on either stack or heap.

3.2 Addressing Memory Protection Schemes

During the last decade, numerous memory protection schemes have been developed and integrated into Linux, in order to make exploitation harder. Most importantly for our purposes, *Address Space Layout Randomization* (ASLR) was integrated (though, in a weak form) in kernel 2.6.12 in 2005 (Gisbert and Ripoll, 2014).

On most Linux systems nowadays, ASLR is enabled by default. This feature can be switched on or off at runtime, using the `/proc` kernel interface. Specifically, the pseudo-file `/proc/sys/kernel/randomize_va_space` may be set to three possible values (Oracle Corp., 2014) to either (i) completely disable ASLR, (ii) to randomize the position of the stack, the virtual dynamic shared object page (VDSO) and shared memory regions, or to use (iii) *Full-ASLR* which additionally randomizes the position of the data segment.

Randomization of memory section offsets is used to make it harder for exploits to work, since important memory addresses (like a function's return address on the stack) cannot be easily calculated in advance. However, the `.text` segment on which the offset-vector technique proposed in section 4 relies, is not affected by ASLR.

`.text` segment randomization cannot be generally enforced on Linux. The reason for this lies in the creation process of ELF binaries. In order to support ASLR with `.text` randomization, a program has to be compiled as *Position independent executable* (PIE), which must therefore be done at compile time. However, PIE is not used on the majority of binaries on the most widely used Linux distributions for a variety of reasons, explained in detail in (Gisbert and Ripoll, 2014) and does therefore not interfere with the proposed technique in the first place.

If, however, an operating system is configured to enable full-ASLR while the final executable is PIE-enabled, it is possible to randomize the `.text` section of running processes. Even in such a situation, we can make use of run-time methods in order to acquire the starting address of the `.text` section, using dynamic code-offset calculation (as done in the proof of concept prototype), outlined in the next section.

3.3 Determining `.text` Section Start- and End-Addresses

Without further protective measures, the starting address of the `.text` section is static, since it always resides in the lowest addresses of the virtual memory space, using a fixed offset (e.g. `0x4000`).

With ASLR and PIE enabled, this offset is randomized. On Linux systems, we can take advantage from the almost exclusively used GNU `binutils`. Therein, the GNU linker `ld` takes care of exporting a global variable `__executable_start`, which determines the very beginning of the `.text` section (The GNU Project, 2015).

The downside of using this approach, however, is limited portability and probably will not work with other compilers and linkers, since `__executable_start` does not comply to POSIX or ANSI. It is still possible to acquire the start of the `.text` section manually, e.g. by using C inline assembly to get the `EIP` value at the start of the `main()` function. Note, that in most executables the `main()` function does not mark the actual start of `.text`, since other code often precedes `main()`, like the `.init` section or the *Procedure Linking Table* (PLT). However, as long as the reference point in the code remains the same (points to the same instruction), it is valid for our purposes.

4 PROPOSED METHODOLOGY

4.1 A Word of Caution

We would like to propose a method suitable for situations as described in Section 1, specifically for situations where either

- user-input for using strong cryptography is not practical for any reason, or
- developers want to make it considerably harder for attackers to extract sensitive information, either via analyzing process memory dumps or performing reverse engineering of application binaries, or
- developers want to make it harder for attackers to modify software, e.g. in order to bypass serial number verification routines.

Note: The proposed mechanism is far from cryptographically secure, and we are well aware that it still *can be exploited*. However, the process of memory exploitation should become considerably harder. The very same applies for many advancements in computer security in general and memory protection schemes in particular: For instance, the introduction of memory protections like ASLR, DEP/NX, ASCII-Armoring and Memory Cookies during the last decade have complicated things for attackers, but do by no means provide unconditional security and exploit prevention. Such schemes simply further complicate attempts towards successful exploitation, and require more specialized, complex and, in parts, error-prone techniques (due to the use of methods like address brute-forcing or heap-spraying to overcome randomization).

4.2 Offset Vectoring

The proposed scheme is built upon the idea of modeling portions of sensitive data, or information required for successful process execution (e.g. credentials, passwords, keys, etc.), by referencing existing machine code byte values already present in the `.text` section of a process, rather than storing it in a regular variable or memory space in the `.data` or `.bss` sections.

For example, many software projects contain information like credentials, database passwords, API keys, etc. Very often, these elements are hard-coded in or directly referenced from the source-code for a variety of reasons. Although this is considered bad programming style, there may be cases where applicable and usable alternatives are sparse. Using offset vectoring, a developer could get rid of plain-text passwords in the source code, by not using a string or character array which holds the password bytes, but by referencing bytes already present in memory upon process execution. The most simple approach to do so, would be to crawl through the process-readable memory and create a list of memory references, pointing to the respective byte representations of the password characters.

However, this approach has major defects. Firstly, memory should be considered to be volatile. Writable memory regions cannot be assumed to remain constant during execution. Moreover, established techniques for memory protection like ASLR will often place certain data on different memory addresses, which renders referencing them for sensitive data unusable. Also, it can not be assured that all byte values required for deconstructing a password (or, even more problematic, a binary bitstring of a key) are actually found in the current process memory space.

We therefore restrict memory referencing to the `.text` segment of a running process only. The `.text` segment is in most cases not affected by ASLR and will remain static, even when a process is stopped and restarted (Oracle Corp., 2014).

Beginning and end addresses of the `.text` section are known either at compile time (without ASLR) or at execution time (with full-ASLR and PIE). For future reference, we will use the letter π for any kind of pointer variables, while π_{ts} is referred to as the start of the text segment, and π_{te} as the end of the text segment. During generation/encoding and regeneration/decoding phases, this segment is iterated bytewise from π_{ts} to π_{te}, and started over once the end has been reached. A pointer π_c refers to the currently addressed byte during `.text` segment iteration, and a variable β is used to temporarily store its content.

While π_c is iterating every byte in the `.text` segment in a circular manner starting with π_{ts}, its value is XOR-ed with the contents of β. Moreover, β may or may not be initialized with one or multiple XOR-ed values from an initialization vector to start from (both variants are discussed in subsection 5.3).

On every iteration, the value of β is XOR-ed with the byte π_c is referencing during the current iteration. The key σ we want to encode (e.g. a cryptographic key, a password or any other byte vector) is split into bytes which are processed in strict sequential order. Once a particular byte σ_i from the key vector has been found (is equal to β after the XOR operation), the search continues for σ_{i+1}.

The number of iterations and XOR operations needed to calculate the current σ_i is temporarily stored as unsigned integer in ω and added to the offset vector ν once found. When all elements in σ have been processed, $|\sigma| = |\nu|$ and ν is returned to the caller.

The process of creating the offset vector ν is given in pseudo-code in Algorithm 1, the corresponding key regeneration code is found in Algorithm 2.

The code listings of Algorithm 1 and 2 represent the most basic form of the offset-vector technique. More advanced features discussed in Section 5 - like the introduction of dynamically adapted initialization vectors - are omitted for readability, but have been included in the proof of concept implementation presented in Section 6.

Algorithm 1: Calculation of the offset vector ν.

```
**Data:** key vector $\sigma$, start offset $\theta$
**Result:** offset vector $\nu$
$\pi_{ts} \leftarrow$ start of .text section;
$\pi_{te} \leftarrow$ end of .text section;
$\pi_c \leftarrow \pi_{ts} + \theta$;
$\beta \leftarrow 0, \omega \leftarrow 0$;
$\nu \leftarrow (0)$;
while $\sigma.length > 0$ do
    $\beta \leftarrow \beta \oplus [\pi_c]$;
    $\pi_c \leftarrow \pi_c + 1$;
    if $\pi_c = \pi_{te}$ then
        $\pi_c \leftarrow \pi_{ts} + \theta$;
    end
    if $\beta \neq \sigma.cur$ then
        $\omega \leftarrow \omega + 1$;
        next;
    end
    $\nu$.push $\omega$;
    $\sigma$.pop;
    $\omega \leftarrow 0$;
end
```

Algorithm 2: Regeneration of the key vector σ.

```
**Data:** offset vector $\nu$,
        start offset $\theta$
**Result:** key vector $\sigma$
$\pi_{ts} \leftarrow$ start of .text section;
$\pi_{te} \leftarrow$ end of .text section;
$\pi_c \leftarrow \pi_{ts} + \theta$;
$\beta \leftarrow 0$;
while $\nu.length > 0$ do
    for $i \leftarrow 0$ to $\nu.pop$ do
        $\beta \leftarrow \beta \oplus [\pi_c]$;
        $\pi_c \leftarrow \pi_c + 1$;
        if $\pi_c = \pi_{te}$ then
            $\pi_c \leftarrow \pi_{ts} + \theta$;
        end
    end
    $\sigma$.push $\beta$;
end
```

5 DISCUSSION & RESULTS

5.1 Limitations

The proposed schemes are neither intended to replace proper handling of key or credentials, nor should they be seen as encouragement to actually store such data in source-code or anywhere else without proper utilization of cryptographic operations.

It is intended solely for scenarios where this is not easily possible, e.g. when requesting decryption keys from the user is no option, or when I/O is generally limited. Without introducing further enhancements (as suggested in the upcoming subsection), the technique can, nonwithstandingly but with considerably higher effort, be reverse engineered.

Also, it increases complexity in deployment and maintenance strategies, as outlined below. Solid integration in the build process is highly recommended. However, once integrated, the entire process can be easily automated with no further human intervention whatsoever.

5.2 On Effectiveness and Purpose

The main advantage of the proposed concept is, that any chunk of information can be represented by working with data already present in the code section mapped into memory. It is therefore not necessary to store the corresponding offset vector in the source-code of the executable and can alternatively be stored or shipped externally, e.g. in text files, key stores or in a database.

Using this technique ensures, that no modifications have been made to the original on-disk binary executable. In such a case, an attempt to decode and reconstruct the informational chunk will almost certainly result in the data being scrambled and therefore unusable.

Modifying an executable can be considered common practice among cracker scenes, aiming at removing, destroying or carefully modifying copy protection measures introduced by the original developers and companies of the applications in question. In its most careful and code-integrity-preserving form, such a copy protection bypassing mechanism can be accomplished by locating the machine code instructions responsible for checking the software's validity (e.g. a serial number), further locating the exact conditional taking care thereof, and replacing this very `je` assembly instruction to a `jne` or vice versa. Many techniques exist accomplishing this goal, but most of them conclude in changing the actual machine opcodes present in the executable.

Changing one single byte will inevitably destroy the integrity of the to-be reconstructed information, since even small changes in a single byte value will result in an avalanche effect for all succeeding bytes, similar to the concept of *Cipher block chaining* (Bellare et al., 1994) or hashing algorithms in cryptography.

The nature of this concept also implies, that every particular version of an executable (e.g. a build, patch-level, etc.) must coercively use a dedicated offset vector in order to work purposefully. This strongly suggests, that when using offset vectoring, it should be integrated directly into the software build process to automatically perform vector calculation.

5.3 Security & Usage

The offset-vector technique can be used for any kind of data, and stored at any place, just like any other data handled during software development. Examples for storing the offset-vector would be in-source, in an external text file or in a database tuple. The best location strongly depends on the specific purpose, the security precautions necessary, and the type of information the technique is applied onto.

For credential keys, it is still not recommended to store the offset-vector directly in the source-code of the application. It will, however, significantly exacerbate the effort necessary to extract the obfuscated information. For example, deobfuscation does require actual reverse engineering of the executable assembly, and is not possible anymore with basic approaches for credential and/or key extraction (e.g. string extraction, determining and reading variable locations, etc.).

In cases where user-input is applicable, it is also possible to make use of an initialization vector for the β variable. Although β is only of C data type `unsigned char` and therefore 8 bits long on almost all architectures and platforms, an initialization vector may hold multiple bytes which will be XOR-ed sequentially before starting the actual regeneration process by reading bytes from $\pi_{ts} + \theta$. Hereby, the initialization vector can represent a user- or authentication-server-provided key necessary for successful regeneration.

Moreover, in order to enhance resistance to memory forensics, the regeneration function (shown in Algorithm 2) should be called immediately before the resulting, regenerated portion of information is required during process execution. It is also of highest importance, to securely wipe the memory space where the deobfuscated information has been placed temporarily – otherwise, the key vector may still reside somewhere in memory.

Even in cases where sensitive information is already externalized to an on-disk file and subsequently encrypted with a user-provided passphrase, offset vectoring can still be used with code-integrity protection in mind, as outlined above.

Moreover, the technique allows choosing an arbitrary starting offset θ, which will be added to `&__executable_start`. Depending on the size of the `.text` section, θ can therefore be used as additional (possibly secret) value necessary for successful compuation.

To enhance secrecy of the information obfuscated using the offset-vector technique, using it in combination with Shamir's Secret Sharing Scheme (SSSS) (Shamir, 1979) technique could be beneficial, where the offset-vector obfuscated information is only one share needed to reconstruct the secret information. Other shares could be, among others, a BIOS/UEFI or peripheral serial number, MAC address, or a share provided by a remote server. Since BIOS/UEFI and other hardware serial numbers encode vendors and usually also target market a deployment package could target, e.g., a specific combination of HW and group of users. A remote server could also generate a secret share based on the request origin, e.g., using its IP address, making such executable valid only for certain locations. A potential attacker would need to invest much more effort to reconstruct a matching environment in order to retrieve the secret. We are planning to explore this direction further in our future work, where a secret rest-share is computed given a secret and a set of pre-existing shares. Only the pre-existing shares together with the rest-share should be able to reconstruct the secret in the same way as SSSS describes. This would significantly improve the secrecy and difficulty of unauthorized access and use of such package and actually provide some cryptographic properties, even though the secret-shares would be publicly reconstructible at some point.

6 PROOF OF CONCEPT

The proposed offset-vector technique has been fully implemented in the C programming language, according to the specifications given during the previous sections. It has been tested on 32- and 64-bit Linux systems, with and without memory protections schemes which may interfere with the correct regeneration of the offset vector.

The source-code for the proof of concept implementation has been licensed under the General Public License version 2, and is planned for public release shortly. The code contains representative functions for both, creating an offset vector for a particular, arbitrarily sized, given chunk of information on the one hand, and recreating that very information based on a given offset vector θ and an optional initialization vector.

Considering production environments, it is recommended not to include both, the obfuscation and the regeneration algorithm into one executable. The most direct way would be to generally include the regeneration function into the software using it, and use the obfuscation mechanism only during the build process in order to create the offset vector. It's in the responsibility of the developers to decide, how the offset vector itself, once created, is handled and stored.

The implementation uses dynamic code-offset calculation as described above, providing support for full-ASLR and PIE.

Experiments have been made with multiple different binaries, ranging from very small `.text` sections (almost exclusively containing the regeneration code, about 2200 bytes of machine code) to rather large `.text` sections (e.g. the Evince PDF viewer with about half a million bytes of machine code in memory) with no significant impact on runtime. The average value of ω (π_c increments per byte in σ) was 272.89, just above the maximum number of values a byte can represent. This difference is due to certain bytes occurring in machine code very often, while other bytes are potentially sparse.

```
 ./text-offset-xor create "MYSECRET"
[INFO]  PIE: 1, DEP: 1, ASLR: 2 (full)
[INFO] .text starts at 0x55f862e776b0
[INFO] .text ends   at 0x55f862e789c5
[DEBUG] [#0001 = 0x4d @ 2612]
[DEBUG] [#0002 = 0x59 @ 223]
[...]
[DEBUG] [#0007 = 0x45 @ 120]
[DEBUG] [#0008 = 0x54 @ 1060]
[INFO] Offset map: 8, 2612, 223, 98,
       168, 143, 66, 120, 1060.
```

Listing 1: The calculation of an offset vector from an executable.

Listing 1 provides the calculation of the correct offset vector for a given binary and a given secret to embed. The secret is given directly via the command line is not necessarily restricted to be an ASCII string.

After generating the offset vector, it can either be embedded into the binary in order to be able to recreate the secret by itself (implicit vector inclusion), or stored somewhere else and provided only after certain conditions are met.

```
./text-offset-xor regen
[INFO]  PIE: 1, DEP: 1, ASLR: 2 (full)
[DEBUG] Recovered byte: 4d
[DEBUG] Recovered byte: 59
[...]
[DEBUG] Recovered byte: 54
Reconstructed ASCII key: <MYSECRET>
```

Listing 2: Reregenating a previously embedded secret with implicit offset vector inclusion.

Listing 2 regenerates the secret using the previously embedded offset vector directly from the executable itself. In this prototype, the offset vector is created based on the offset vector calculation binary itself. It can, of course, be also caculated for any other executable.

7 CONCLUSION AND FUTURE WORK

We have presented a scheme for generating predefined chunks of arbitrary information of variable length via the actual machine code of a running process using an *offset vector*. While offset vectoring is no silver bullet for the common key-storage problem of applications, it can be used under certain situations to improve exploitability resistance, hide the information in question from appearing in an executable and, if applied correctly, the process memory map.

It does not interfere with modern memory protection techniques, can be easily automated and integrated in an existing build process, and can help to ensure the integrity of shipped binaries. This can also enhance the resistance against attempts to crack software, or extract valuable information thereof.

Augmenting the existing prototype with an implementation of *Shamir's secret sharing scheme* (Shamir, 1979) could enable a multi-part architecture where the recoverable secret is only one of several different parts needed.

Moreover, the offset vector itself does not necessarily need to be included in the executable directly. It can also be utilized as an authentication vector only given to the client once certaion conditions are met (e.g. payment has been successfully done).

Another approach would be to scatter multiple offset vectors across different binaries (executables and software libraries) of a given software product in order to generate the final secret whereas the integrity of all different files must therefore be intact. Once again, considering this approach, the Shamir secret sharing scheme would be suitable in such a way that a certain number of binaries must be present and unaltered in order to regenerate a secret.

The given implementation has only been developed to work on x86 systems using the Linux operating system. Other than that, future work may include an implementation for several other architectures, whereas especially ARM and MIPS seem of interest due to the high number of devices being built on these architectures on the one hand, and the fact that many of them show limits regarding user interaction (e.g. home routers and WiFi access points). Further development regarding other operating systems is also considered to be a favorable goal in the near future.

Considering multiple binaries running simulaneously and communicating with each other over a network connection, a valid offset vector and reregenated secret could be used as starting point for mutual trust without user interaction.

ACKNOWLEDGEMENTS

This research paper and all corresponding source code developments are a result of the *Research Group for Secure and Intelligent Mobile Systems* (SIMS).

REFERENCES

Bellare, M., Kilian, J., and Rogaway, P. (1994). The security of cipher block chaining. In *Annual International Cryptology Conference*, pages 341–358. Springer.

CERN Computer Security Team (2015). How to keep secrets secret. alternatives to hardcoding passwords. CERN.

Erickson, J. (2007). Hacking - the art of exploitation 2nd ed. No Starch Press.

Gisbert, H. and Ripoll, I. (2014). On the effectiveness of full-aslr on 64-bit linux. International In-Depth Security Conference Europe (DeepSeC), Vienna.

Henriksen, M. (2015). Gitrob: Putting the Open Source in OSINT. xx.

Ligh, M., Case, A., Levy, L., and Walters, A. (2014). The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory. John Wiley & Sons.

Naaktgeboren, A. (2015). Change the expiration of master password probes to never expire. Mozilla Firefox Source Code, changeset `edd395471638`.

Oracle Corp. (2014). Oracle linux security guide for release 6 – configuring and using kernel security mechanisms. Oracle.

Prandini, M. and Ramilli, M. (2012). Return-oriented programming. IEEE Computing Society, Journal for Security & Privacy, Vol. 10, No. 6.

Samyde, D., Skorobogatov, S., Anderson, R., and J., Q. (2002). On a new way to read data from memory. First International IEEE Security in Storage Workshop.

Shamir, A. (1979). How to share a secret. Commun. ACM, Vol. 22, No. 11, pp. 612–613.

The GNU Project (2015). Gnu linker ld documentation (gnu binutils) version 2.25. GNU.

Zhenghong, W. and Ruby, B. (2007). New cache designs for thwarting software cache-based side channel attacks. Proceedings of the 34th Annual International Symposium on Computer Architecture, New York.